lecture 2

1. 16个register 和 xPSR（execution program status ）
   - 。r13 stack–pointer
   - 。r14 linker–pointer
   - 。r15 program counter
2. 内存中各部分的位置和大小

3 little endian 和 big endian： 每一个variable整体都是从小的地址往大的地址写。 但是在每个variable内部， big endian是越靠左的bit在越小的地址。

4 什么是directive： 指令，但是该种指令用于assembler或者compiler。其本身并不是executable instruction

5 命令：

MOVT： 只改变register中top的16个bit

ldrb： 往32 bit的register里load最低的8位，剩下的补充0

ldrsb：往32 bit的register里load最低的8位，剩下的按照第八位实行带符号的补偿

6 instruction中出现的addressing mode的区别

offset addressing： 就是在base register的基础上加上offset的值，然后得到最终的目标地址 [<Rn>, <offset>]

pre–indexed addressing： 从base address 加 offset那里load数据，但是将offset加到base address之上 [<Rn>, <offset>]!

post–indexed addressing： 从base address 那里load数据，在load结束之后，再将offset加到base address 上 [<Rn>], <offset>

```
data:
    .byte 0x12, 20, 0x20, -1
func:
    mov r0, #0
    mov r4, #0
    movw   r1, #:lower16:data
    movt   r1, #:upper16:data
top:
    ldrb   r2, [r1],#1
    add r4, r4, r2
    add r0, r0, #1
    cmp r0, #4
    bne top
```

7 instruction如何encoding 有16 bit 和32 bit两种mode

8 branch instruction

B BX BL BLX CBNZ CBZ TBB TBH

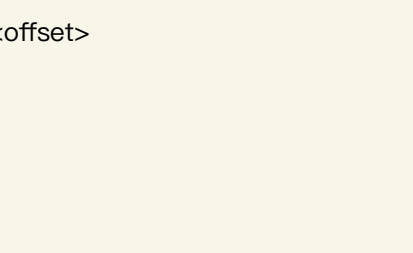
9 data processing instruction

10 load 和 store的不同instruction

11 miscellaneous instruction 杂项指令

12 如何在现有指令的基础上，更新APSR的状态： ADD–》ADDS

13 condition execution中都包含哪些condition



14 ABI： 即ISA所遵守的默认规则

- 如 R9: static base register
- 如 R10: stack limit checking 若SP超过这个register中存储的值，则视为stack overflow
- 如 R12: scratch register 通常被用作一个draft note， 该数据可以随时被新数据所覆盖
- 如 R13: stack pointer
- 如 R14: link register 每当我们branchdao le 一个sub function并且运行玩sub function之后，我们要branch 回到 R14 中所存储的地址
- 如 R15: program counter
- 如 R7: frame pointer： 用来point to current stack frame in a function。

简单来说： R0 – R3 需要caller save 其他的除了R12 都需要callee save

15 build process： assamble files 先compile成object file， 然后将object file 用linker链接起来成为execution file 然后运行

16 nested function 从 C 转化成assambly code 的例子：

```
#include "func1.h"

int func2(int x) {
        func1(5);
        return x * 3;
}
```

```
func2.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <func2>:
#include "func1.h"

int func2(int x) {
   0:   b580            push    {r7, lr}
   2:   b082            sub     sp, #8
   4:   af00            add     r7, sp, #0
   6:   6078            str     r0, [r7, #4]
        func1(5);
   8:   2005            movs    r0, #5
   a:   f7ff fffe       bl      0 <func1>
        return x * 3;
   e:   687a            ldr     r2, [r7, #4]
  10:   4613            mov     r3, r2
  12:   005b            lsls    r3, r3, #1
  14:   4413            add     r3, r2
}
  16:   4618            mov     r0, r3
  18:   3708            adds    r7, #8
  1a:   46bd            mov     sp, r7
  1c:   bd80            pop     {r7, pc}
```

lecture 3:

1 logic gates 复习：

注意什么是 bubble pushing

2 memory–mapped I/O：

1 旧的I/O的缺点是，我们需要另外的instruction 用来read 和 write to peripheral devices。 但是这是资源的浪费因为我们已经有了load和store instruction

2 新的方法： 将peripheral 放在memory address上，也就是说某一个区间的地址对应特定的peripheral device， 那么当我们需要读取或者修改peripheral device时，直接访问对应的地址即可。

3 bus acces中所涉及的部分：

initiator： 负责在总线程上发送命令请求的设备， 可以发送读取或者书写指令

target：负责会用initiator所发出的命令：可以是准备从initiator手中接收某个值，或者向initiator提供某个值

4 bus access 中所用到的信号名称和作用：

REQ#：request —》当该信号是0的时候表示initiator发出了命令请求，1时则表示initiator当前没有请求

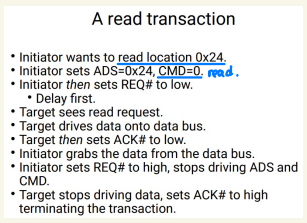
CMD：command —〉当该信号是0 的时候表示read指令， 1时表示是write的指令

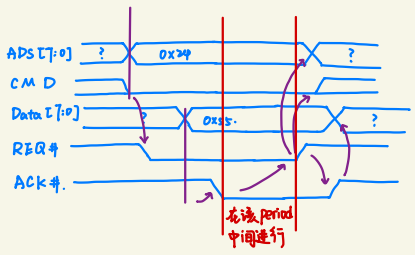
ACK#： acknowledgement —》当该信号是0时表示peripheral已经处理好了数据，此时initiator就可以通过bus得到相关的数据，当该信号时1时表示peripheral还没有准备好，需要initiator继续等待。

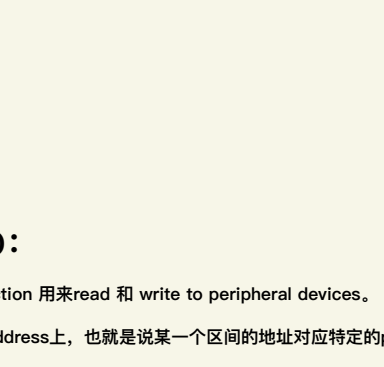
Data[7:0]: 用于传输data

ADS[7:0]：用于明确地址，明确要将哪一个peripheral当成此次transmission的target

最理想的状态是：initiator将ADS， CMD， 和 REQ 同时setup好。—》但是产生的问题是因为这个过程是asynchronous的，所以无法保证三个信号能在同时setup好，所以我们选择delay REQ信号变化的时间，确保地址和指令的类型已经setup 好之后再发出REQ的指令。







抉择：什么时候让ACK#从0 变到 1 进而告知initiatorper peripheral已经完成了数据的write —》 1. 直到完成才改变， 2. 将data放在local queue

lecture 4

1 DMA： direct memory access， 其好处是不同peripheral可以相互之间交流，而不用再通过process。

之前：如果要从peripheral A 读取一个数据到peripheral B，那么就需要processor先向A发出read请求，之后再向B发送write请求。

现在：processor告诉DMA controller 从A读取数据到B，然后DMA controller 就回去执行该命令，但是processor就能去做其他事情。

2 AHB 和 APB 之间的配合：

首先：在AHB上发起一个transmission request：比如从某个memory location 读取数据，再写到另一个memory location去。

其次：APB发现相关的地址是属于他掌管的peripheral的，就会将 read 和 write指令转移到APB上，并且让链接在起上的peripheral执行。：

3 bus 中的术语： 什么是initiator， 什么是target

4 wire的连接分类：

多个peripheral share同一个数据线：使用tri–state 或者 open– collector 的方式保证每次只有一个设备在控制wire

每个peripheral 由自己单独的数据线：

5 APB 的设计特点：

low–bandwidth: 数据传输量小

non– piplined：一个transmission只有在上一个已经完成的情况下才能进行。

6 APB的bus signals：

shared：

PADDR：bus上的address

PWDATA：从processor向peripheral 写入一个数据

PWRITE：0 —》表示read 1 —〉 表示write

PCLOCK：控制的clock

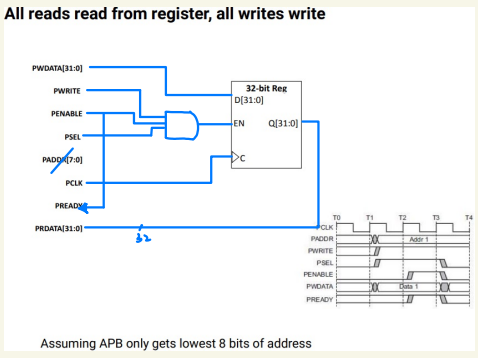
PENABLE：1 —》表示initiator正在bus上处理一个transmission。

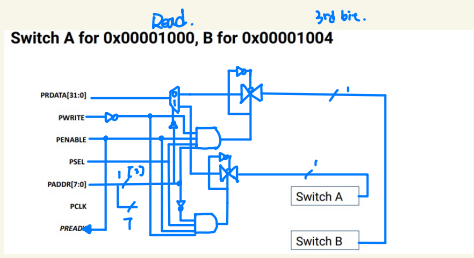
individual：

PSEL：1 —》 表示改peripheral是本次transmission的target

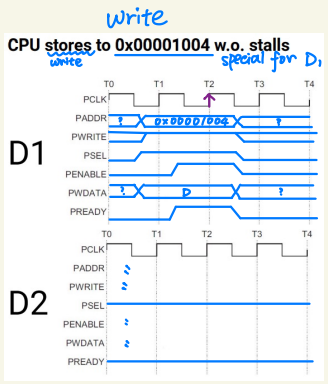
PREADY：1 —〉表示 peripheral已经做好准备接收，或被读取。

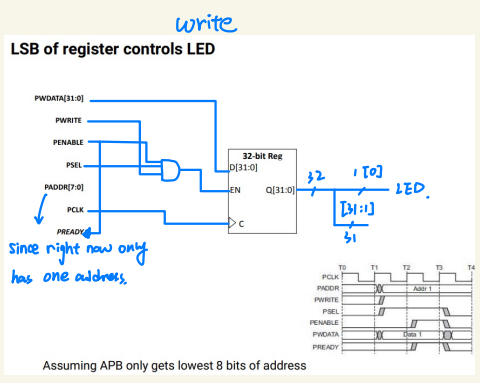
PRDATA：从peripheral 向processor 读取一个数据

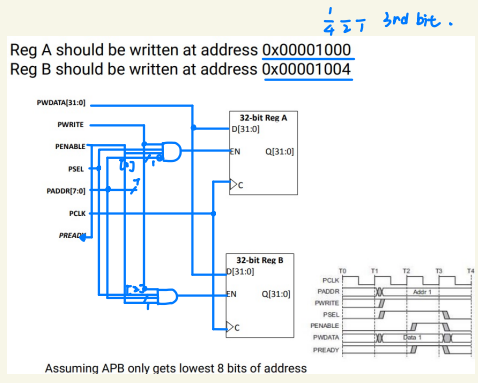


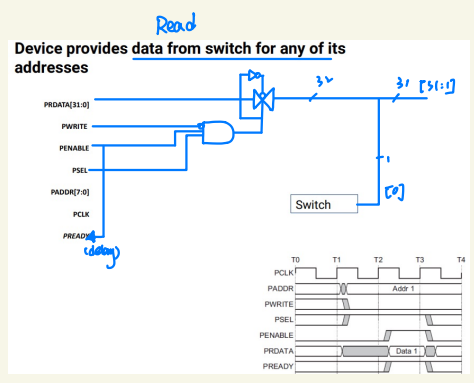


7 例子：









8 wait state： 目的是为了给peripheral be ready 预留时间

首先setup state：PADDR，PSEL，PWRITE 的value set up好

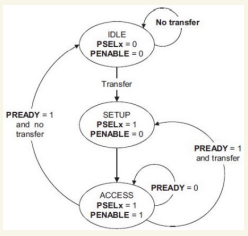
其次在第一个wait state：PENABLE 变为1， 表示现在在bus上将要进行transmission。

同时PREADY变成0，迫使所有其他信号进入等待，因为peripheralt 仍需要时间准备

紧接着在第二个wait state中：所有信号不发生变化，即继续等待peripheral进行准备

最终在access state中，PREADY 从0 变到 1， 表示peripheral准备就绪。在下一个raising edge的时候，transmission完成。

9 APB state machine的变化和条件：



lecture5

1 pointer to 未知的数据类型（在run的时候才被确定下来）

语法：void* x

作为parameter input到function中： int apple_checker (const void *x);

2 function pointer 即将function作为parameter input到另一个function当中

语法：int check_stuff(void *stuff array, int (*checker)(const void*));

check_stuff：一个function output是int。 output是一个装有 void* pointer的array 和 一个function pointer。

例子：

int (*checker)(const void*)：首先：*checker 一》“*” 表明这是一个pointer， checker是function pointer的名字

其次：“int” 表明这个function的output是int，

最后：“const void*” 表明这个function的input是一个指向未知数据类型的pointer，并且我们不能通过这个pointer修改所指对象的值

用作定义jump table：即一个装有function pointer的array：

int (*func_ptr[3]) (const void*) = {func1, func2, func3};

在该例子中，func_ptr是一个装有三个function pointer的array，并且这些function的input都是const void* 且output都是int

但有时在同一个array中装的function pointer 并不是同一个input 和 output类型：

解决方法：void (*func_ptr[3]) = {func1, func2, func3};

3 interupt：

1》CPU知道什么时候data 准备就是的方式：

polling： while loop不断询问

interrupt：中断主程序 + 执行ISR （interrupt handler / service routine）+ resume 到主程序

2》interrupt的分类：

instruction interrupt：由于软件程序运行不当引起的，比如 /0， 比如unimplemented instruction

external interrupt: 由外部因素引起的，比如press button， 比如电源中断

3》如何知道进入哪个interrupt：

a。 如果知道interrupt发起的来源：

1）按照发起的source在interrupt vector table中找到jump table里的index

2）按照得到的index在jump table中找到对应的interrupt service routine的address

3）跳到ISR的位置，执行interrupt的内容

4）之后在根据 link register中的address跳回之前的主线程

b。如果不知道interrupt发起的来源：

1）就要把所有的resource都遍历一遍

4》怎么知道ISR结束之后回哪里：

a。提出user mode 和 system mode的概念：

1）没有interrupt出现的时候是user mode， 当interrupt出现是转变成system mode

2）为保证之前的在主程序中的所有state不会丢失，system mode会copy所有的register，并且在system mode期间使用shadow registers。 当返回主程序之后再删掉shadow registers

4》interrupt相关的信号：

a。interrupt request：当hardware发起interrupt时变成1，但一段时间之后又会变成0.

b。interrupt pending status：软件在检测到硬件发起了interrupt request的时候就会挂起interrupt pending state。此时即使interrupt request变为0也不影响。

c。processsor mode：当interrupt pending state 变成1时，他就开始从thread mode 变成handler mode。当processor mode 变化之后，interrupt pending state的任务就完成，就能变成0了。

d。interrupt active status：当ISR的handler执行的时候，active status就会变成1，当ISR return的时候，interrupt active status就会变成0.

5》tail chaining:

用于解决当一个interrupt正在执行，又有一个新的interrupt进入的情况：避免在system mode和user mode中来回切换，避免了对于register里面的值的copy和restore。

6》interrupt 有三个相关的configuration需要设置：

a。interrupt set enable 和 interrupt clear enable

b。interrupt set pending 和 interrupt clear pending

c。interrupt active status register

7》interrupt的priority问题：

a。分为preempt priority 和 subset priority：

1）当两个interrupt不同时间出现的时候：

当他们preempt相同时：先出现的先执行，后出现的等到先出现的完全执行完再开始执行

当他们preempt不同是：若后出现的preempt更小，则先出现的interrupt被终止，直到后出现的interrupt执行完后再继续执行

2）当两个interrupt相同时间出现的时候：

当他们的preempt相同时：比较subset priority，越小的越早执行，大的要等大小的执行完后再开始执行。

当他们的preempt不同时：preempt越小的越先开始执行，待到小的执行完之后，大的再开始执行。

8》用于筛选interrupt的masks：

a。PRIMASK：可以set成0或1，当set成1时，除了NMI 和 hardfault 以外的所有interrupt全部被无视

1）NMI：non-Mashable-interrupt: 比如断电，系统重启这类优先级非常高的interrupt

2）hardfault：通常是软件程序在运行过程中遇到的优先级很高的interrupt—》比如访问invalid address，比如stack overflow

b。FAULTMAKS：可以set成0或1，当set成1时，除了NMI以外的所有interrupt全部被无视

c。BASEPRI：base priority mask register —》set它一个数值，比这个数值低的优先级的interrupt会被忽略掉。